BBN Systems and Technologies Corporation

A Subsidiary of Bolt Beranek and Newman Inc.

(2)

AD-A210 238

Report No. 6972

Tools for Application Development in Heterogeneous Distributed Environments

G. Falk, K. Anderson, M. Thome, M. Dean and T. Reinhardt

March 29, 1989

Prepared by:

BBN Systems and Technologies Corporation
10 Moulton Street
Cambridge, MA 02138

Submitted to:

Defense Advanced Research Projects Agency
Arlington, Virginia

BBN Systems and Technologies Corporation

A Subsidiary of Bolt Beranek and Newman Inc.

Report No. 6972

Tools for Application Development in Heterogeneous Distributed Environments

G. Falk, K. Anderson, M. Thome, M. Dean and T. Reinhardt

March 29, 1989

Prepared by:

BBN Systems and Technologies Corporation
10 Moulton Street
Cambridge, MA 02138

Submitted to:

Defense Advanced Research Projects Agency
Arlington, Virginia

Report No. 6972

Tools for Application Development in Heterogeneous Distributed Environments

G. Falk, K. Anderson, M. Thome, M. Dean and T. Reinhardt

March 29, 1989

Prepared by:

BBN Systems and Technologies Corporation
10 Moulton Street
Cambridge, MA 02138

Submitted to:

Defense Advanced Research Projects Agency
Arlington, Virginia

Report No. 6972

Tools for Aplication Development in Heterogeneous Distributed Environments

G. Falk, K. Anderson, M. Thome, M. Dean and T. Reinhardt

March 29, 1989

Prepared By:

BBN Systems and Technologies Corporation
10 Moulton Street
Cambridge, MA 02138

Prepared For:

Defense Advanced Research Projects Agency
Arlington, Virginia

Contents

Abstract

Military software systems based on distributed heterogeneous computing resources are being deployed at an increasing rate. Existing networks, distributed operating systems, and software tools provide the basic elements of a system substrate for applications development in such environments. To improve programmer productivity, however, it is necessary to enhance these existing facilities with higher-level programming constructs oriented toward concurrent programming in the large. The Tool for Large-grain Concurrency (TLC), described in this report, provides a basic set of such capabilities. TLC is based on COMMON LISP and CLOS (*the Common Lisp* Object System), translates high-level functional programs into operations of the underlying systems, and coordinates concurrent execution of these subsystems on distributed heterogeneous resources. The usefulness of TLC has been evaluated in the domain of the CASES decision support system of the Fleet Command Center Battle Management Program.

1. Background

1.1 Needs

This work represents an important start, a feasibility demonstration, toward solving two problems confronting the next generation of distributed systems:

- *A general environment for concurrency* —exploiting the non-overlapping strengths of various parallel computer hardware architectures; and,
- *A general environment for heterogeneous architectures* —providing an effective software environment for heterogeneous, distributed software and hardware.

Concurrency in the Large

The next generation of distributed systems offers the opportunity to capitalize on diverse parallel computer architectures. Presently, the various classes of hardware architectures (e.g., connection machines, vector processors, and medium-grained MIMD machines) are appropriate primarily for complementary classes of algorithms. Typical C^3I systems will involve a mix from all of those classes of algorithms.

Networking these various parallel processors together is already within the state of the art. What is needed is an effective means of distributing program segments to the computer architecture best suited to the algorithms represented by each. Moreover, this should be accomplished with minimal burden on the person needing the results of the program. Further automation of the process of distributing code to the appropriate processor will minimize programmer burden, which is, at present, substantial in order to use any parallel processor.

Exploiting Heterogeneity

The opportunity to introduce new hardware and software systems in a military context normally requires not only its coexistence with existing hardware/software systems, but also an ability to work in concert with them. Networked, heterogeneous systems will be the norm, we believe, not the exception. Networking such systems is not a problem. What is needed is a software environment that facilitates effective use both by the end user and the programmer.

The problem in complex distributed applications is to provide a software infrastructure which

1. supports distribution of program segments to the most suitable parallel processor;
2. insulates the developer from the idiosyncrasies of the underlying systems,;
3. facilitates effective use of the available underlying systems in carrying out higher level application functions; and
4. provides a modular expandable base for integrating new functionality, hardware, and software.

1.2 Application Context

We have begun prototyping the type of software needed in a system called Tools for Large-Grained Concurrency (TLC) in the context of problems arising in DARPA's Fleet Command Center Battle Mangagement Program (FCCBMP). The FCCBMP environment is installed at the Pacific Fleet Command Center, Pearl Harbor, Hawaii, and includes five major hardware technologies (VAX, Symbolics, Encore multiprocessor, SUN, and Butterfly™ multiprocessor), five different operating systems (VMS, Genera, UMAX, UNIX, and Chrysalis/Mach), application subsystems including Oracle, KnowledgeCraft, KEE, FRESH, IRUS, OSGP/DMDS, and numerous Navy engagement and environmental models. These application subsystems are written in several different languages and use different representations for basic data types.

We have focussed on a particular application in the FCCBMP, the CASES expert system. CASES provides Navy users with facilities for evaluating the relative warfighting capabilities of US and USSR forces. Both the need for parallelism and the need to integrate many heterogeneous subsystems arise in CASES. Concurrency is necessary because near real-time response is critical. Since the algorithms CASES uses span a broad range of types, no single parallel computer is ideal for all of the algorithms; a range of parallel computer architectures is important.

CASES also requires access to subsystems running heterogenous hardware and software. No common substrate exists at the operating systems level; therefore alternatives, such as MACH, do not apply. Figure 1.1 illustrates some of the major subsystems of the CASES decision support system in the FCCBMP at CINCPACFLT, which is used as an illustrative example throughout this document. CASES provides Navy users with facilities for evaluating the relative warfighting capabilities of US and USSR forces. The subsystems in the top half of the figure provide services to a Symbolics workstation which is, at present, both the primary development environment and end user environment for CASES.

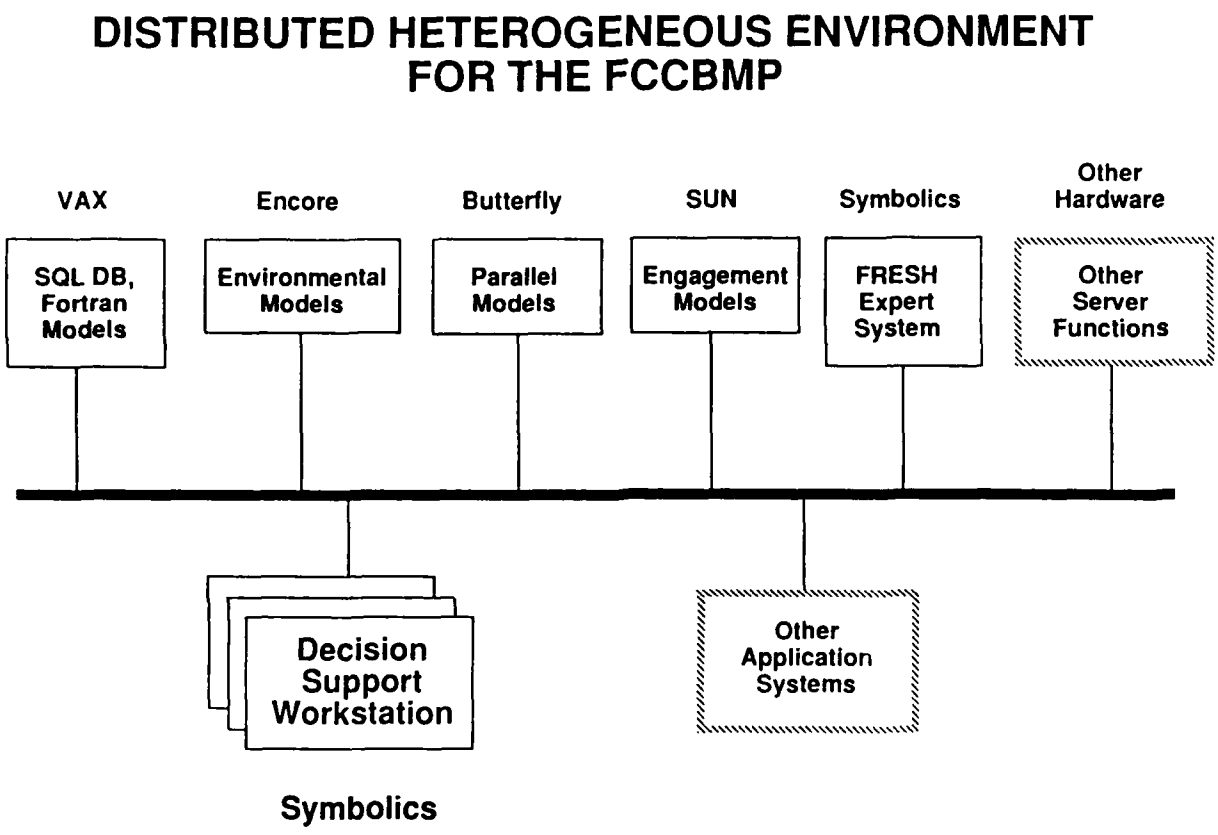


Figure 1.1: CASES subsystems.

1.3 Our Approach

In our approach, a distributed operating system (e.g., CRONUS [SVB*88], running on top of MACH) provides communication capability among heterogeneous hardware/software systems. Only a small CRONUS module need be written for each combination of hardware and resident operating system. TLC then adds to the distributed operating system the ability to translate a declarative program into processes for the various subsystems.

In either a multiprocessor or multiple computer environment, the application developer needs a convenient means to specify which portions of a computation can proceed concurrently. The run-time system needs to be able to initiate appropriate subcomputations that proceed in parallel as well as coordinate the processing of asynchronously received subsystem responses.

In a critical decision support environment such as CASES, new technology (e.g. database servers, massively parallel processors) can speed the decision making process and potentially improve it by providing the time to consider more alternatives. Thus, allowing for new technologies symbiotically with existing capabilities is another important characteristic of large application systems. Such systems evolve over time as new functionality and performance demands are placed on them. Additional copies of existing servers as well as new servers must be accommodated. More user interface workstations may be added to a system that originally supported only single user operation. Finally, in developing a particular application, one must be aware of potential requirements to support future applications that rely on the same set of underlying servers.

Part of the desired infrastructure can be provided by existing technology: Operating systems on the constituent hardware provide the mechanisms for managing local resources. An Ethernet and standard communication protocols (e.g. TCP/IP) provide mechanisms for reliably passing uninterpreted bits among constituent subsystems. A distributed operating system, such as CRONUS, can be run on top of this communications substrate to provide basic support for distributed resource management, remote procedure calls, and format conversion between heterogeneous data representations. What is not available at present is an application programming substrate which facilitates concurrent programming in the large. This is the the goal of TLC.

The focus of the TLC system is large grain concurrency as illustrated in Figure 1.2. TLC facilitates parallel programming by providing a high-level functional programming interface to the user. The TLC system translates this declarative program specficiation into the appropriate "forks" and "joins" without the programmer having to become involved at the level of synchronizing individual processes.

PARALLELISM IN COMPUTER SYSTEMS

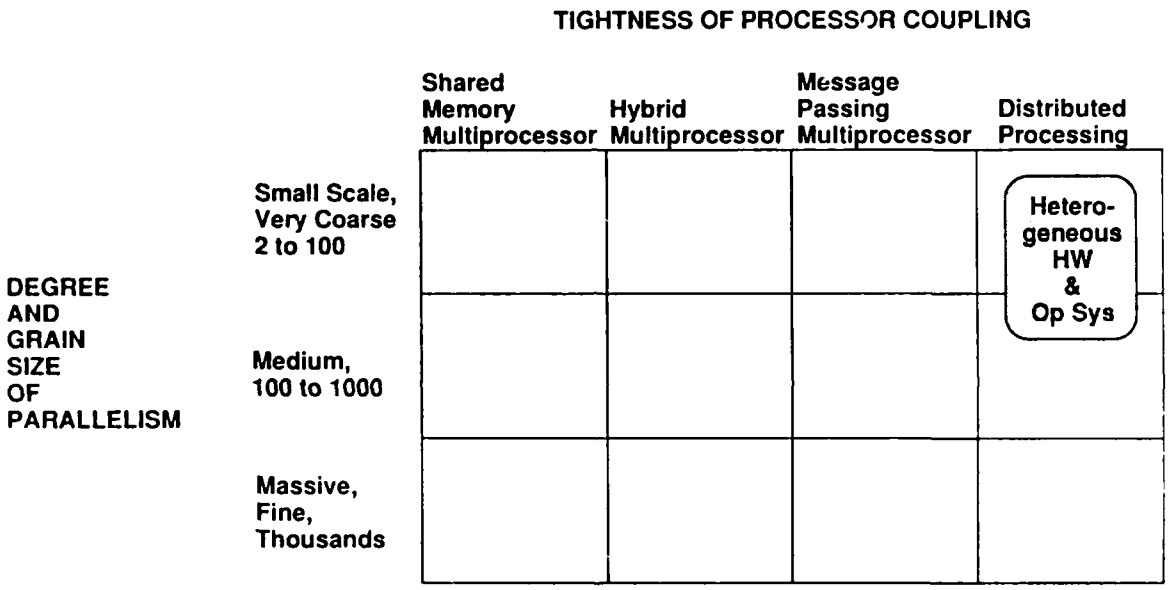


Figure 1.2: TLC Focus.

2. System Overview

This chapter presents an overview of the TLC system. First, a typical problem of the type the system is designed to address is presented. Second, an overview of the TLC system architecture which supports that functionality is described.

2.1 An Example

In order to better understand the problem and the tools, it may be helpful to examine a concrete example. Consider the following query, which is typical of those posed to the CASES system:

> Which of the [currently displayed] submarines has the greatest probability of locating SPA[1] 1 within 10 hours?

To answer this query, CASES uses a Navy model called SPASEARCH. SPASEARCH determines the probability of a particular submarine locating a particular SPA within N hours. As illustrated in Figure 2.1, the SPASEARCH model (simplified for the purpose of this exposition) takes three input parameters, **Total-time**, **Travel-time**, and **Search-rate**, and runs on the Butterfly multiprocessor.

These inputs can be manually supplied or can be provided as the outputs of additional module runs. Figure 2.1 illustrates the composition of the primitive modules which can be applied to evaluate SPASEARCH for each of the currently displayed submarines. In the figure, modules are indicated by boxes and data accesses are indicated without boxes. Some of the data accesses may be to local information and others may be references to databases and knowledge bases residing on remote computers. Similarly, some of the procedures are carried out locally and others need to be run on other processors.

For a given submarine, many of the computations and data accesses are independent and can be carried out concurrently. For example, the "Time Late" and "Search Rate" computations and their subcomputations can be carried out in parallel. Similarly, all three data accesses required by SWEEP could be be concurrent operations. TLC eases the load

[1] SPA stands for SOSUS Probability Area and represents an area of uncertainty in which a detected enemy unit is located.

EXAMPLE PROBLEM/QUERY

"Which of the [currently displayed] submarines has the largest probability of locating SPA A within 10 hours?"

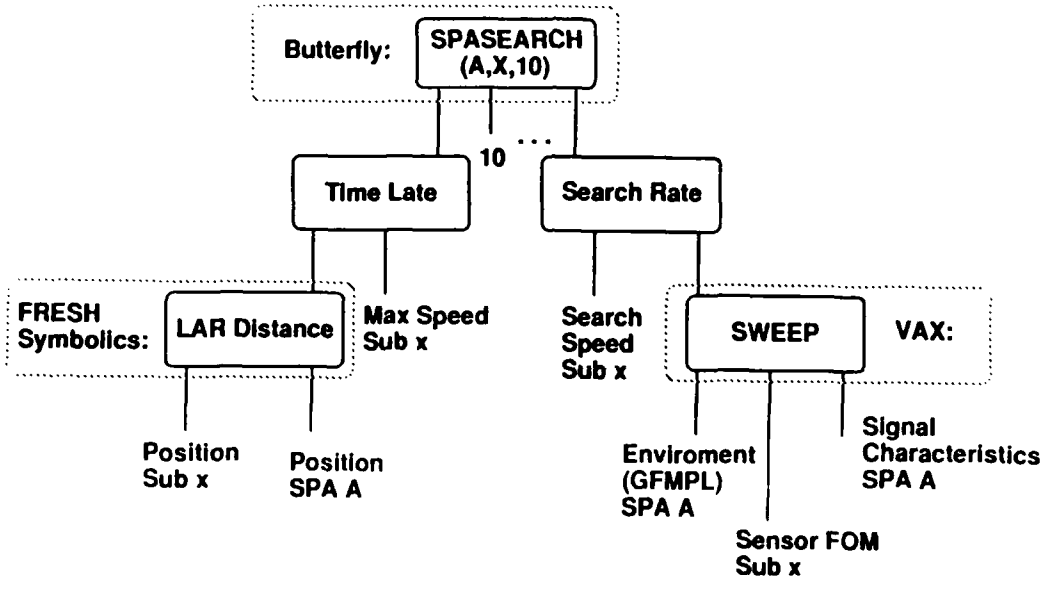


Figure 2.1: Decomposition of SPASEARCH Model.

on a programmer seeking concurrency. The programmer can realize this concurrency described by evaluating CONCURRENT-SPASEARCH defined as follows using TLC:[2]

```
(DefFunction CONCURRENT-SPASEARCH(a x z)
  (SPASEARCH (TIME-LATE
              (LAR-DISTANCE
               (POSITION x)
               (POSITION a)))
             (SPEED x)
             (SEARCH-RATE
              (SEARCH-SPEED x)
              (SWEEP
               (ENVIRONMENT-FACTOR a)
               (SENSOR-FIGURE-OF-MERIT x)
               (SIGNAL a)))
             Z))
```

2.2 System Architecture

The TLC system consists of three parts:

1. a language for concurrent functional programming,
2. a concurrent virtual machine (VM) supported by heterogeneous distributed systems integrated via CRONUS, and
3. a translator for compiling TLC functional programs into VM code.

These three components are described in Chapters 3, 4, and Appendix A, respectively.

Figure 2.2 illustrates the integration of TLC into the CASES environment. A CRONUS kernel is installed on each of the constituent systems. This kernel runs as an application process and coordinates all CRONUS activity on that host. In addition, a set of CRONUS Object Managers coordinate interactions with local servers modules (e.g. database servers, expert systems).

CRONUS operates on top of TCP/IP, which handles basic reliable transmission. CRONUS implements canonical data types and provides remote procedure call communication between applications running on different systems. The TLC VM fields requests for service from the user and determines if the service can be supported locally or requires a remote invocation via CRONUS. The TLC VM also enhances the CRONUS RPC mechanism to support concurrent invocations (multiple outstanding requests).

The TLC translator compiles user written functional programs into code which runs on the TLC VM (see Figure 2.3). This translation is a two stage process. The first phase of

[2] Chapter 3, page 12, below, provides a complete description of the TLC language and its semantics. Also, Chapter 4.3.2 contains a discussion of data level pipelining by the addition of *streams*.

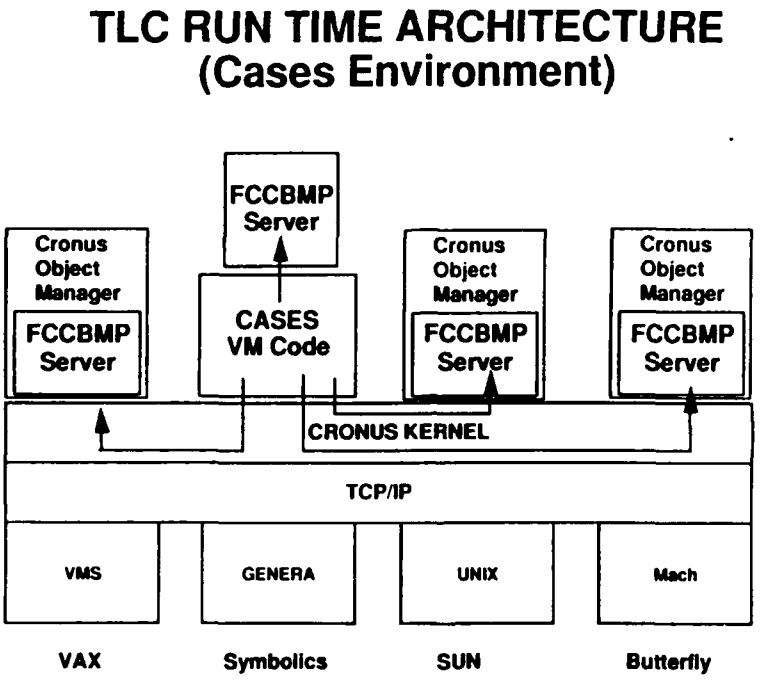


Figure 2.2: TLC Integration in CASES Environment.

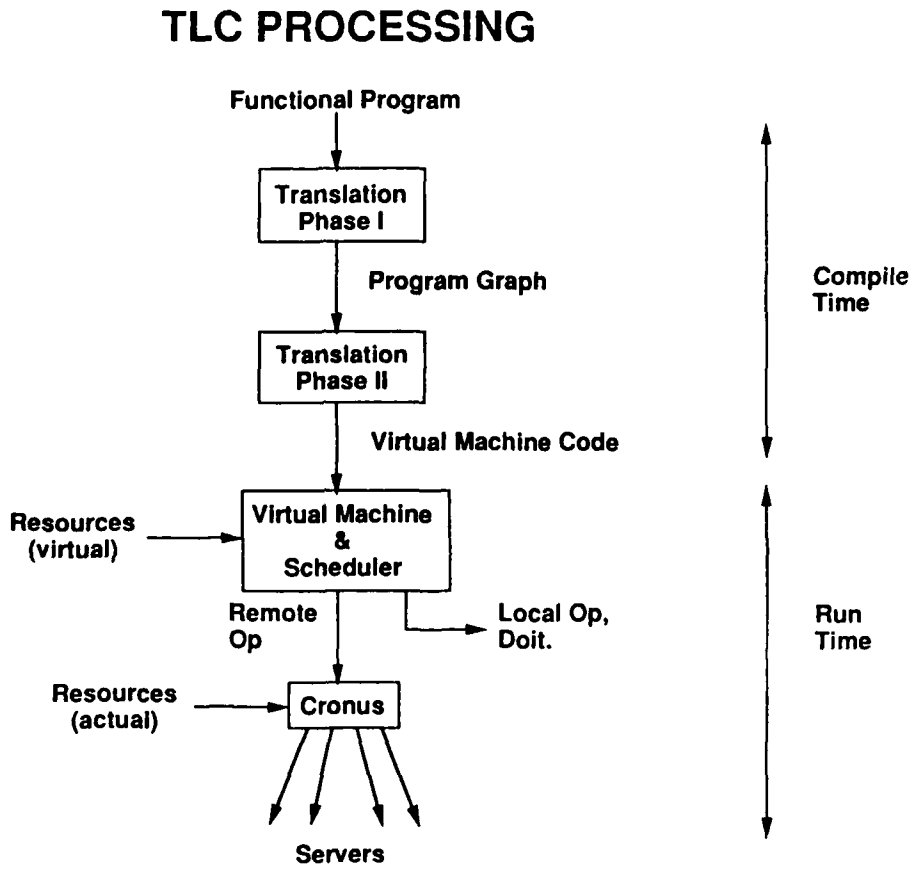


Figure 2.3: TLC Translation Process.

the translation maps the source text into an internally represented directed acyclic program graph (DAG). The DAG explicitly represents the concurrency within the program. This portion of the translation process is independent of the target VM. The second phase of the translation process takes the DAG as input and produces VM code that can be executed. This portion of the translator is dependent on the target VM. The VM that currently exists is written in COMMON LISP. It is possible to envision a VM that is based on the C language although no effort in this direction currently exists.

The current TLC VM operates in a staightforward manner: It simply executes local operations directly and passes any non-local operation to CRONUS for remote execution. It incorporates neither complex resource management mechanisms nor any higher level functionalities, such as a high level description system in which to observe its own behavior or a planning system with which to interact with its environment.

In the future, we anticipate that the VM will be augmented with such mechanisms to ultimately support a knowledge-based scheduler that will use problem domain specific knowledge and run time information (parameter values, machine loading, etc.) to improve system performance and utilization of resources. For example, the VM code might indicate that the operation SWEEP is to be carried out. Assume that there are two versions of SWEEP, SWEEP-SERIAL and SWEEP-PARALLEL, which run on uniprocessors and multiprocessors, respectively. Based on the argument(s) to SWEEP, the intelligent scheduler would determine whether SWEEP-SERIAL is adequate or SWEEP-PARALLEL is more appropriate. Assuming that SWEEP-PARALLEL is needed, CRONUS would determine which of the available hardware resources capable of running SWEEP-PARALLEL would be used as the actual server. Additional information on the knowledge-based scheduler is contained in Chapter 4.3.2, on page 36.

3. The TLC Language

3.1 Overview

TLC is an extensible language that supports applications development in dynamically evolving, heterogeneous environments. Underlying its design is the recognition that problem solving in such open environments requires a language that

- expresses and exploits the concurrency intrinsic to such systems; and,
- is easily extended to express a variety of algorithmic as well as organizational needs.

The current TLC compiler accepts as input a functional specification in the form of an s-expression and produces as output a directed acyclic graph (a DAG) which is interpreted by the TLC virtual machine and executed across the network. While the DAG represents a maximally concurrent decomposition of a solution, it is the actual status of the network and the virtual machine's priorities at the time of execution that determines how much concurrency is realized. TLC comprises three distinct layers, each addressing various needs:

The Applications Layer which is of interest to applications developers. Here is where applications level functions are defined in terms of other high level functions and organizational issues, such as resource allocation strategies, are implemented;

The Language Development Layer where language designers can extend the TLC language in terms of compiler primitives; and,

The Primitive Layer which is where new constructs that differ from the overall semantics of the language, and hence cannot be expressed in terms of the existing primitives, are implemented.

In the rest of this section, we will concentrate on the applications level, postponing detailed discussion of the language development and primitive layers to Appendix A, page 41.

3.2 The Applications Layer

On the surface, TLC syntax is a subset of the syntax of COMMON LISP [Ste84]. It differs in its fundamental semantics and in its size, however. COMMON LISP is much larger and specifies a sequential semantics. TLC, on the other hand, is *fundamentally concurrent*; sequential execution is regarded as a restricted, degenerate case of concurrency. For instance, COMMON LISP semantics specifies that **fn2** and **fn3**, below, are evaluated left to right, sequentially:

```
(fn1 (fn2 arg-1 arg-2) (fn3 arg-1 arg-2))
```

TLC, however, specifies that **fn2** and **fn3** are evaluated *concurrently*: That is, **fn2** might precede **fn3**, may follow it, or may be evaluated at the same time, i.e., in parallel. Concurrent evaluation is subject to two types of restrictions:

1. *Implicit restrictions* that originate out of data dependencies. For instance,

   ```
   (fn1 (fn2 arg-1 arg-2) arg-3)
   ```

   restricts the evaluation of **fn1** to occur after the evaluation of **fn2** —although the evaluation of **fn2** may proceed concurrently with the evaluation of **arg-3**; and,

2. *Explicit restrictions* that are imposed by the programmer by employing a sequential special form (discussed below) for doing so.

3.2.1 Command and Expression Contexts in TLC

A "purely functional" language provides *expression context only*: it assumes that the evaluation of forms are localized phenomena that always return a value. The original LISP 1.0 [McC60], based on the λ-calculus [Chu41], is an example of such a language; it provided no semantics or linguistic support for state change operations [Bac78]. [1] What it did provide, however, was an intrinsically concurrent evaluation formalism for expressing purely functional notions. λ-expressions are intrinsically concurrent —we cannot deduce the order in which they were evaluated based upon the results of their evaluation.

Because TLC is a general-purpose applications development tool, however, it must provide support for tractably expressing state change operations. This means that we are faced with the prospects of tempering the large degree of intrinsic concurrency with serialization points around state change operators.

[1] ... although proponents of pure functional programming have long been aware of this shortcoming and have proposed and implemented various stream-based, feedback mechanisms.

Rather than employ a global process-locking scheme, similar to a *monitor* abstraction, TLC distinguishes between *command* and *expression* contexts. Expressions are TLC forms that, as a result of their evaluation, return a value. The COMMON LISP function, **list**, is an example of an expression. Invoking **list** on a number of arguments returns a list of those arguments. It causes no effects outside of its invocation and returns a predictable result: It returns a *new list* with each call.

Commands, on the other hand, do not return "dependable values" as a result of their evaluation, but instead, cause some "effect." The COMMON LISP function, **rplaca**, is an example of a command: It destructively replaces the head of its first argument (which should be a list) with the second argument.

Hence, depending upon whether a form appears within expression or command context, different degrees of concurrency might be appropriate. Consider the following example, where the three subexpressions are evaluated concurrently and the value of the last, i.e., the **(first x)**, is returned as the result:

```
( ... (list x y)
      (rplaca x y)
      (first x))
```

list is an expression, i.e., it does not modify either **x** or **y**, but the **rplaca** does —it destructively replaces the head of list **x** with **y**. What is actually returned as a result of executing this code fragment is problematic and depends upon how TLC interprets such "mixed forms." It remains an open question, at this writing, how to best optimize mixed forms, i.e., combinations of commands and expressions, for concurrent evaluation. Some discussion of these issues is found in Chapter 4.

3.2.2 Input to TLC

The TLC parser accepts a valid s-expression as input[2] and subjects it to the following parse rules:

- Strings, arrays and **quote**'ed forms are treated as constant data, and are passed through;
- Atoms are either numbers or symbols:
  - Numbers are passed through;
  - Symbols are examined for symbol-macro definitions. If none are found, they are passed through, otherwise the symbol macro definition is expanded in place, and the parser is recursively invoked on the result.

[2] That is, what the Common Lisp reader accepts is considered valid input.

- Lists with atomic first elements are interpreted as function calls where the head of the list is the function spec and the tail its arguments. Function specs may have the following definitions:
  - As a TLC macro, in which case retrieve the macro definition and recursively parse its expansion, in place;
  - As a TLC expression or command, in which case retrieve its parser, apply it to the input stream and continue; or,
  - As a generic function call. This means that it is either something that has a definition as a TLC function, i.e., it is a TLC lambda form, or it is a COMMON LISP function. In either event, the reference is left inline and will be resolved at execution time. Its argument list, however, will be parsed so as to be evaluated concurrently.
- Finally, when encountering a list of lists in the body of a special form, such as a **let**,

```
(let ((...))
  (fn1 arg-1 arg-2)
  (fn2 arg-1 arg-2) ... )
```

  parse them all so that they are evaluated concurrently with the value of the last being returned as the value of evaluating the entire list of forms.

As an example, consider a subcomputation of the **concurrent-spa-search**, introduced on page 8, in the previous chapter:

```
(lar-distance (position ::) (position a))
```

is parsed such that the subexpressions **(position x)** and **(position a)** are evaluated concurrently and the **lar-distance** operation is performed at the "join" —as indicated in Figure 3.1.

As a somewhat artificial example of multiple forms occuring within the body of a **let** special form, consider

```
(if (predicate a b)
    (let ()
      (then-function-1 a)
      (then-function-2 b)
      t)
    (let ()
      (else-function-1 b)
      (else-function-2 a)
      nil))
```

which results in the DAG in Figure 3.2, below.

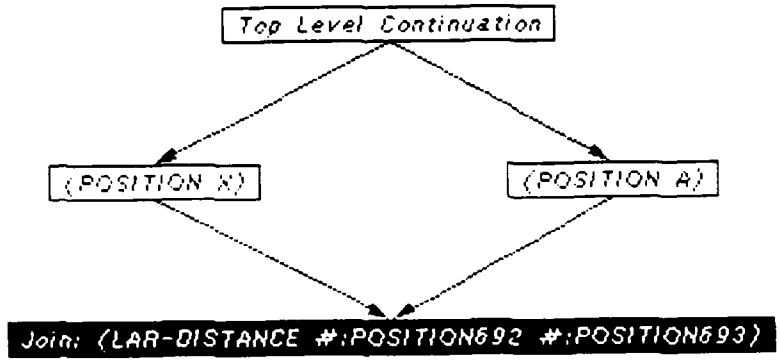


Figure 3.1: DAG of a SPASEARCH subcomputation, (**LAR-DISTANCE** (**POSITION X**) (**POSITION A**)).

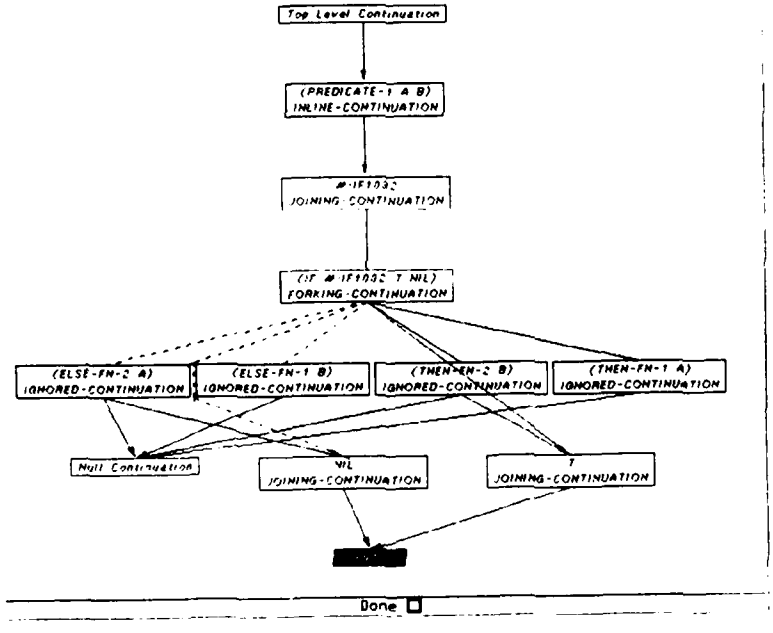


Figure 3.2: DAG of a conditional with concurrent then and else clauses whose results are ignored (as indicated by the dashed lines).

3.3 Application Level Primitives

The following primitives are available at the applications level: **defname**, **deffunction**, **defprocedure**, **defmacro**, for defining top level forms, **lambda**, **named-lambda**, **command-lambda**, **named-command-lambda**, for defining intermediate behaviors, and **let**, **let***, **if**, **and**, **and***, **or**, **or*** and **sequential**. All, except **defname**, **deffunction**, **defprocedure**, **defmacro** and the various **lambda** forms, were defined at the language development level, see Appendix A.1, in terms of the **dlet** and **dif** TLC primitives described in Appendix A.3.1, page 42.

3.3.1 Augmenting the Top Level Environment

Because TLC is lexically scoped, **let** forms may only be used to associate symbols with definitions in local environments. A **defname** special form is provided for associating definitions with symbols at the top level. Its syntax is:

```
DEFNAME          :: (defname <id> <expression>)
<id>             :: <symbol>
<expression>     :: <symbol> | <list>
```

and its semantics is to evaluate its <expression> in the top level environment and associate its <id> with the value returned from the evaluation. **defname** is similar to the COMMON LISP form, **defvar**, which is used to define "free" variables, i.e., identifiers at the top level.

3.3.2 Defining Functional Forms

The **deffunction** and **defprocedure** forms are provided for defining functions and procedures, respectively, at the top level. Functions are defined as entities that, upon evaluation, return a value. Procedures, on the other hand, do not return "dependable" values, but are instead evaluated for their "effects."

Functional forms are used for expressing modular elements of algorithms —much as they would be in a functional subset of COMMON LISP. Recursion is the default control structure, as of this writing. The familiar **Factorial** function is represented:

```
(DefFunction Factorial (n)
  (if (= n 0)
      1
      (* n (factorial (1- n)))))
```

for instance. Alternatively, this could have been expressed in terms of **defname** and **named-lambda** forms:

```
(DefName Factorial
  (named-lambda Factorial (n)
    (if (= n 0) ...)))
```

Note, the "free operators," i.e., =, * and 1-, are the COMMON LISP functions, left inline. Their argument lists, however, have been lifted so that any available concurrency is exploited. In this particular example, however, all operators are either unary or are consist of simple terms, e.g., (= **n** 0). Even the (* **n** (**factorial** (1- **n**))) form is done sequentially because **n** is simple and is therefore passed through.

Using Lambda Forms to Define Behaviors

Unlike COMMON LISP, TLC supports command as well as expression contexts, hence four, instead of two, lambda forms exist: **command-lambda** and **named-command-lambda** — which are the command context analogs to **lambda** and **named-lambda**. In addition, TLC's binding scheme supports local behavior definitions, using the lambda forms, without special enclosing forms. For instance, in COMMON LISP a **label** or **flet** is required to associate symbols with behaviors in a local binding context, whereas any of the let forms in TLC are acceptable. This means that TLC symbols are dereferenced as functions or values and not both, as in COMMON LISP.

TLC **lambda** expressions are lexically scoped and are first class objects. Their syntax is similar to the COMMON LISP **lambda**:

```
LAMBDA-EXPRESSION   :: (<lambda-form> <lambda-arglist> <body>)
<lambda-form>       :: lambda | named-lambda | command-lambda
                       named-command-lambda
<lambda-arglist>    :: (<symbol>*)
<body>              :: <expression>*
```

but, the entire range of lambda list variables supported by COMMON LISP are not yet supported by TLC. The semantics of lambda expressions is similar in many ways to **let** (described below): The arguments in the <lambda-arglist> are evaluated and bound concurrently, and the <expression>'s that compise the <body> are evaluated concurrently with the last being the value returned.

3.3.3 TLC For Decomposing Intermediate Forms

Intermediate forms are statements that are not enclosed in top level forms, such as a **defname** or **deffunction**. Examples of intermediate forms include valid s-expressions, as in our earlier example of summing a product and difference, see page 15, above. Because the current TLC compiler isn't "closed" in the sense of accepting only TLC primitives,

i.e., COMMON LISP function calls are left inline and their arguments, if left unqualified, are lifted, intermediate level forms may also be compiled.[3]

The remaining primitives are the basic building blocks of intermediate forms.

Using Let For Controlling Evaluation

Two **let** forms exist for expressing a range of evaluation strategies. They have the same syntax which is similiar to COMMON LISP:

```
LET             :: (<let-form> <let-arm>* <body>)
<let-arm>       :: (<let-var> <expression>) | ()
<let-form>      :: LET | LET*
<let-var>       :: <symbol>
<body>          :: <expression>*
<expression>    :: <symbol> | <list>
```

and the semantics for each form is:

let evaluates and binds its `<let-arm>`'s concurrently; and,

let* evaluates and binds its `<let-arm>`'s sequentially.

Each guarantees the availability of its `<let-var>`'s *before* processing the `<body>`. In this respect, they are similar to the QLISP [GM88] construct, **qlet**, except for the case where **eager** is the value of its guard:

```
(qlet NIL <let-arms>* <body>)
```

is equivalent to **let***, and

```
(qlet T <let-arms>* <body>)
```

is equivalent to **let**. With the addition of *eager evaluation* constructs, e.g., **futures** [BH77] processing, the complete semantics of **qlet**, and a host of other, interesting binding constructs, such as **letrec**, a recursive let form, become available.

[3] But see page 21, for a discussion of controlling the evaluation of argument lists to take into account flow of control issues.

Using If to Control Flow

The syntax and semantics of **if** is similar to the COMMON LISP special form of the same name:

```
IF                :: (if <pred> <then-clause> <else-clause>)
<pred>            :: <expression>
<then-clause>     :: <expression>
<else-clause>     :: <expression>
```

if evaluates its <pred> *before* either its <then-clause> or <else-clause>.

Combinations of If and Let

Various degrees of concurrency can be expressed by combining **let** and **if** forms. For instance, using **let** expressions as the <then-clause> and <else-clause> yields an **if** expression with concurrent then and else clauses:

```
(if <pred>
  (let ((ignore (then-clause-1-function ... ))
        (ignore (then-clause-2-function ...))
        ...
        (then-value (then-clause-n-function ...)))
    then-value)
  (let ((ignore (else-clause-1-function ... ))
        (ignore (else-clause-2-function ...))
        ...
        (else-value (else-clause-n-function ...)))
    else-value))
```

The TLC **defmacro** form could be used to capture this cliché as a macro definition, **cIF** (for concurrent IF), with the following syntax:

```
cIF               :: (cif <pred>
                          (then <then-clauses>)
                          (else <else-clauses>))
<pred>            :: <expression>
<then-clauses>    :: <expression>*
<else-clauses>    :: <expression>*
```

and with the following macro definition:

```
(defmacro cIF (predicate then-clauses else-clauses)
  (let ((then-value (last then-clauses))
        (else-value (last else-clauses)))
    `(if ,predicate
       (let ,@(mapcar #'(lambda (then-clause)
                          `(ignore ,then-clause))
                      (butlast then-clauses)))
       (let ,@(mapcar #'(lambda (else-clause)
                          `(ignore ,else-clause))
                      (butlast else-clauses))))))
```

Alternatively, this new definition could be bootstrapped at the level of compiler primitives, see Appendix A.1.

3.3.4 Expressing Different Types of Concurrency

Consider the common problem of selecting some element, x, from a potentially large universe, $\mathcal{U}$, such that $P(x)$. This is the sort of problem that many hope concurrency will aid in solving. Depending upon the certain criteria, however, concurrency may or may not offer much advantage over sequential processing.

For instance, if the problem is treated as purely set-theoretic, and no constaints are placed upon the order of evaluation of the elements of $\mathcal{U}$, then an obvious solution would be to write:

```
(or (p x1) (p x2) ... )
```

knowing that TLC will specify that the subexpressions, i.e., the $(p\ x_i)$'s, be evaluated concurrently. Upon their completion, then, the first non-empty element of the resulting set is selected.

Such a solution is wasteful and the question naturally arises: If we're asked for an element, x, why are we creating the set of x such that $P(x)$, rather than just finding an instance? The maximally concurrent approach of evaluating each $(p\ x_i)$ is appropriate only if the application requires "all x, such that ..." which is not the case.

Protocols do exist, moreover, for controlling concurrency and the mechanisms for introducing them are being added to TLC as of this writing, see Chapter 4.3.2, page 36. The idea here being to distribute resource management with the computation. In this way, numerous subcomputations may be initiatiated and, upon the fulfillment of some condition, the remaining subcomputations may be *gracefully aborted*. In addition, various type of concurrency, e.g., *greedy* and *cooperative*, may likewise be implemented: Some discussion of these issues is found on page 30, Chapter 4.2.10.

Alternately, an argument for COMMON LISP's sequential semantics is attractive in the case where only *one candidate*, x, is required *and* the evaluation of the $(p\ x_i)$'s has side

effects.[4] TLC recognizes the need for this type of serialization, when "flow of control" issues must be taken into account, and provides two compiler level primitives, **and*** and **or***, as sequential analogs to **and** and **or**, see Appendix A.1.1, page 42.

3.3.5 Summary: The Current State of TLC

TLC provides a high level representation within which various types of concurrency may be expressed. The TLC compiler translates this s-expression representation into a DAG, and it is the responsibility of the VM layer to "interpret" or "execute" this DAG: This means facilitating computation across the network of available machines, *eventually* taking into account the specific issues of *migrating* data in heterogeneous networks, distributing *resource management*, and *load balancing* —i.e., distributing work among a community of specialized as well as general-purpose architectures.

3.3.6 The Current Implementation

Presently, the TLC VM is based upon COMMON LISP closures, augmented with the CRONUS instruction set. Many of the fundamental issues, such as migration and load balancing, are possible only in the context of a more "comprehensive" design, see chapter 4.2.1, page 23. Nevertheless, the current ad hoc scheme demonstrates the utility of simple communications between heterogeneous computional agents in a potentially dynamic topology.

3.3.7 Translating TLC

As expected, translation from functionally concurrent trees (which is represented by the DAG) into nested closures is straightforward and reasonably efficient.

The current implementation includes the primitives and behaviors outlined above. It is written in COMMON LISP and runs on Symbolics 3600 workstations, running Genera 7.2.

[4] In large, open-ended systems, reversing an effect might be intractable since the resource is shared by many processes.

4. Future Work

Although TLC can be used in its present state to facilitate concurrent programming in the large, it is most appropriately viewed as a prototype. There are several directions in which work on the TLC prototype should proceed. We can categorize these directions as

- applications,
- enhancements,
- and integration.

described in the remainder of this section.

4.1 Applications

Our goal has been to develop a tool that will be useful to application programmers. CASES has been the application domain which has motivated the work to date. We hope to integrate TLC with the CASES software development environment and incrementally improve TLC functionality based on actual development activities. We also hope to identify additional development environments and apply TLC in these domains as well.

4.2 Basic Enhancements

There are number of areas where we already know TLC requires additional work. These areas are identified and described briefly below.

4.2.1 Need For A More Appropriate Architecture

Most existing architectures are based upon the von Neumann model of computation and are inappropriate foundations upon which to support computation in distributed, heterogeneous systems which are fundamentally open-ended and evolutionary in character. The current virtual machine layer, described at the conclusion of the previous chapter, is inadequate in several respects:

- It does not provide a uniform, extensible substrate upon which to construct more expressive communications protocols;
- It provides no specific mechanisms for addressing the issues of *migration* of data and *load balancing* of work across machines; and,
- It provides little support for error handling and debugging.

—to name a few. In response to these, and other issues, a more integrated architecture, called to TLC VM, is under development.

4.2.2 The TLC VM is Class Based

Written in COMMON LISP, the VM is a CLOS-based system that implements the necessary primitives to "bootstrap" TLC programs onto a particular machine.

At the VM level, data is viewed as instances of classes of data. For instance, the number 4 is a particular instance of the class *fixnum*. Hence, various behaviors, i.e., methods, may be attached to this instance in an extensible, tractable manner —that is, through the CLOS **defclass** and **defmethod** facility. Such class-based systems are portable, fundamentally extensible and desirable from an open systems perspective.

4.2.3 Shared vs. Local Memory Models

Traditional wisdom has it that programmers think in terms of "shared" memory ... doubtless attributed to their von Neumann roots. The approach taken in the design of the TLC VM is that the programmer's view of memory should be *virtual* and *indifferent* to the underlying architecture. In the most flexible scheme, one where data is migrated freely between machines, the programmer cannot know at runtime the location of particular data. Even in the more restricted case where data is asserted to be available on a particular machine, for reasons of robustness, redundancy must be taken into account.

Nevertheless, for reasons of maintaining *arms-length interactions* between applications and enforcing machine-specific data dependencies, a facility must be made available for programmers to assert and maintain that a *class of data* is local to a particular machine. In the current effort, CLOS classes and methods provide this functionality. Such locality or non-migratability is maintained at the VM level as part of an abstract data type's class behavior.

For instance, requests to migrate numbers are handled differently than requests to migrate machine-specific data, such as hash-tables. In the first case, migration is permitted and the number is packaged in an acceptable manner to CRONUS and migrated to the requesting host. In the second case, however, the hash-table is not migrated and the **migrate** method, instead, insures that accesses to it are handled remotely —that is, the requests are migrated to the machine hosting the hash-table.

4.2.4 The Structure of the VM

Computation is Task-Based

The fundamental unit of execution is the *task* which is an ordered pair: $\langle T, S \rangle$, where T is some *target* object which is an instance of a predefined class, and S is a *specification* which is an object incorporating a method call and any additional arguments that is being "sent to" or "invoked upon" the target, T.

Hence, each task could be thought of as a transmittable closure which is queued for execution on any of the various *dispatchers*. Any number of dispatchers may exist on a given machine. Their behaviors are as follows: Upon a clock tick, or by any other means available to a particular architecture, dequeue the next task. If there is none, ask the surrounding dispatchers for spare tasks, i.e., tasks on their work queues that are migratable to this dispatcher.

Assuming that a task exists, decompose it into its target and specification. If the target is local, i.e., is an instance of a local data type, such as numbers, invoke the method-name from the specification on this instance, passing any additional arguments specified in the task through to the method call. Otherwise, remotely invoke the method via CRONUS.

A Closer Look at a Task's Components

A task's target can be either a local, COMMON LISP data type or a remote TLC reference which is represented as an instance of the **mailbox** class. Mailboxes provide a uniform unit of addressability at the virtual machine level —pointers perform a similar role in traditional computer languages. Specifically, mailboxes have *resident* data, a *queue* of incoming requests for that data and a *forward-to* slot which is used by the migration machinery.

Specifications embody information required by the target to perform some action, i.e., the name of the local method to invoke and additional data with which to invoke it. Specifications have types as well, they are either *requests*, *replies* or *complaints*. Request specifications include a

message-name which is a name of the function to be invoked on the target,

args which are additional arguments that are passed to the target;

continuation which is an object that represents the "rest of the computation." In essence, it is the proxy for the remaining computation and is the object to which any subsequent replies are sent, and a

resource-mgr which is an object that provides funds for the computation to proceed.

Hence, requests most closely resemble function calls. For each request, moreover, at least one reply (or complaint) will issue. Reply specifications are similar to requests except that they contain no continuations —they are sent to continuations in response to the request. Finally, complaint specifications resemble replies in the current implementation except that their contents are messages about exceptional conditions and *become* for all intent and purpose the value of the remaining computation and are passed "up the chain." Hence, in the absence of an application to "trap" and "handle" the exception, the error is returned and any necessary systems bookkeeping is performed.

4.2.5 The VM Language

Unlike the TLC language it supports, the VM language is comprised almost entirely of *commands*, i.e., constructs that, as a result of their evaluation, cause some effect and do not return dependable values. The only expression defined at the VM level is *create*, which is the VM primitive for creating instances of VM level objects. The VM command language provides, among other things, the architectural support to effect function calling in distributed environments where the actual function calling mechanism is distributed and is better thought of as a common communications protocol of such systems.

Primitives for Supporting Function Invocation

The **request** command has the following syntax:

```
REQUEST          :: (request <target> <msg-name> <args>
                       :continuation <continuation>
                       :resource-mgr <resource-mgr>
                       :key <reply-key>)
<target>         :: <cronus-address> | <local-reference>
<msg-name>       :: <symbol>
<reply-key>      :: <cronus-token>
<args>           :: <symbol>*
```

and is interpreted by the VM to mean that a request for some processing, specified by the <msg-name>, is to be communicated to the <target>. The <resource-mgr> is responsible for providing computational currency and the reply is to be directed to the <continuation>. The <reply-key> is currently unused, but will be used to disambiguate multiple replies.

If the <target> is a <local-reference>, i.e., it is on the *dispatcher*'s machine, this is just a funcall of the form:

```
(funcall <msg-name> <target> <args>)
```

whose result is wrapped in a **reply** command and sent to the <continuation>. Otherwise, a CRONUS remote procedure occurs and the results are likewise transmitted to the <continuation>.

<continuation>'s embody the remainder of the computation: Each contains an *environment* which provides a context within which its *form* is to be executed. Transmitting a result to a continuation invokes its *form* on the incoming message, within whatever context has been established by its *environment* and causes control to pass to its *siblings*.[1]

reply-to commands are similar, syntactically, to **requests** except they require no continuations (although the <target> is usually a continuation) and no <msg-name> because continuations are created with their own local states that embody the original **request**'s msg-name:

```
REPLY          :: (reply-to <target> <value>
                     :resource-manager <resource-mgr>
                     :reply-key <cronus-token>)
```

As a result of its evaluation, the **reply** command causes the <value> to be transmitted to the <target>, providing that sufficient resources are available through the <resource-mgr>. If the target is local, the <target> is funcalled with the <value> as the argument list, otherwise a CRONUS invocation transmits the <value> to the <target>. The <reply-key> is used to disambiguate multiple replies directed to the same continuation.

Reply-to commands, together with **requests**, are sufficient to capture the notion of function calling or subroutine evaluation in standard computer languages. Although not yet supported, they enable a function to return multiple values via the use of <reply-key>'s. They also integrate resource management in a de-coupled, yet extensible manner.

Handling Exceptional Conditions

Besides the anticipated reply, exceptional conditions arise and require some mechanism to ensure system robustness. To this end, the **complain-to** command exists. It has the same syntax as **reply-to**. Its semantics is somewhat different, however. All continuations are created with some "default" exception handling code, thus the reception of a complaint, unless otherwise provided for, causes the object that received the complaint to return a standard message with the keyword, :aborted, as the first element and with the specific exception as the rest of the message. In addition, any local bookkeepping, such as **mailbox** *queue* management, is performed at this point.

[1] Because continuations embody local environments, *history sensitive* computations can be expressed as localized phenomena, i.e., with a continuation embodying each slice. This must be compared and contrasted with the more global, von Neumann notions of free variables which would quickly become a communications bottleneck in the general case.

Again, for generality's sake, **complain-to**'s behavior is implemented as a CLOS method specializable for particular classes of objects. Hence, machine designers can tailor exception handling to include arbitrarily complex behaviors.

4.2.6 Create Provides VM Level Instantiation

The only expression provided at the VM level is **create**; its syntax is:

```
CREATE              :: (create <id> :args <arg-pairs>
                        :resource-mgr <resource-mgr>)
<arg-pairs>         :: (<keyword> <symbol>)*
```

and its sematics is to create an instance of class of object denoted by `<id>` and to initialize it accorrding to the `<arg-pairs>`. The `<resource-mrg>` is checked for necessary funds before the process is begun. From the user's perspective, **create** is atomic.

create is not used to make instances of builtin COMMON LISP data types such as strings or fixnums. Instead, it is used to create instances of VM specific classes: For instance, programmers might need to define a new class of data, such as migratable hash-tables, for a particular application. This provides a logical place to bootstrap such functionality within the mechanism of the VM.

4.2.7 Primitives for Supporting Communications Protocols

Function invocation, as expressed in terms of **request** and **reply** commands, constitutes a simple communication protocol. Other, more subtle and complex protocols are needed, however. For instance, some primitive VM level commands are necessary to support communications forwarding. When a reference is found to be non-local or is determined to have been *migrated* for reasons of locality, for instance, some protocol has to be in place for ensuring that the communication finds its target. To this end, a **forward-to** primitive exists:

```
FORWARD-TO          :: (forward-to <cronus-address> <task>)
<task>              :: <task-object>
```

Its semantics is to transmit the `<task>`, as an opaque object, to the `<cronus-address>`

4.2.8 Primtives to Support Delegation

Delegation is a communication protocol whereby an object may receive a message and *delegate* some or all of the processing to a group of objects in a *managerial* or *supervisory*

manner. This makes it possible to describe complex behaviors, such as inheritance and recursive invocations on an object that is locked. Hence, a delegation protocol, where a specification may be received by the target and delegated, in whole or in part, is desirable. A VM primitive, **delgate**, must be added with the following syntax and semantics:

```
DELEGATE        :: (delegate <client> <msg>)
<client>        :: <cronus-address>
<msg>           :: (<msg-name> <args>)
<msg-name>      :: <id>
<args>          :: <atom*>
```

delegate sends a <msg> to its <client> in such a manner that the reply from the <client> *is not queued* on the delegatee. That is, message delivery is tightly coupled, resembling a phone-call as opposed to a mail delivery.

4.2.9 Primitives to Support State Change

The **update** and **replace** commands provide localized and global state change, respectively.

```
UPDATE          :: (update <cronus-address> <index> <value>)
<index>         :: integer
```

and,

```
REPLACE         :: (replace <cronus-address> <cronus-address>)
```

express two very different notions:

In the first case, **update**, the <cronus-address> and the object that it references remains unchanged. That is, if <cronus-address> points to an object, $\mathcal{O}_i$, after executing the **update** instruction it points to the same object, $\mathcal{O}_i$, except that some state change *local to* $\mathcal{O}_i$ has been effected.

In the case of **replace**, however, the object dereferenced by the <cronus-address> *is different*. In some sense, this is semantically equivalent to changing the value-cell of a free variable with the COMMON LISP **setq** construct.

For debuggability, previous references, in the case of **replace** instructions, may be saved away in order to provide some reconstructive history keeping mechanism. Hence, communications arriving at a **replaced** object *follow its* **forward-to** *links* as a matter of course. Just how long these chains can grow without impacting upon storage and performance is an open question and good fodder for future research.

Maintaining a history of **update** instructions is more difficult: Unlike **replace** the reference remains unchanged from an outside observer's perspective. Perhaps some method of saving the history of communications processed on this object is feasible —in this way browsing the order of **update** commands reveals the order of state changes on this object.

4.2.10 Primitives for Managing Resources

That the <resource-mgr> is a required component of each command that effects processing, viz. **request**, **reply-to** and **complain-to**, or storage, **create**, provides a powerful resource control tool. Because these <resource-mgr>'s are instances of a class of objects, called the **resource-manager-class**, their behaviors are likewise specializable and extensible. Although the default behavior for instances of **resource-manager-class** is primitive, as of this writing, arbitrarily complex behaviors are possible. Consider the following potential scenarios:

Competitive Concurrency

Here numerous tasks are spawned concurrently. Each <resource-mgr> spawns sibling managers and allocates a portion of its computational resources to its siblings. Siblings allocate their resources recursively[2] and must ask their immediate parents for more when they are depleted.

The "top level" resource manager is controlled by a "managerial process" that reviews the work undertaken on each of its immediate branches. Hence, a competitive environment is set up where immediate siblings may be rewarded with more processing resource depending upon the top level manager. Upon sucessful either completion or abandonment of the top level task, resources are cut off via the top level resource manager and the sibling computations, upon subsequent requests for more funds, are gracefully aborted.

Cooperative Concurrency

Here a similar situation is established as in the case for competitive concurrency, except that *sibling nodes share resources* through interaction with their parents.

For example, two tasks are concurrently spawned to perform a computation which requires partitioning a large set into mutually exclusive subsets. In this case, candidates failing one criterium pass the other. Moreoever, if these are computationally expensive predicates, it is more efficient for one sibling task to "reward" its counterpart with a percentage of its computational resources upon receipt of a valid candiate.

[2]That is, each sibling becomes a parent when requested to spawn subtasks, and so on.

4.2.11 The Translation Process

The TLC compiler produces a DAG whose nodes are *continuations*. Continuations are objects that embody *environments* which provide a context and *form*'s, i.e., segments of code to be executed within that context. Moreover, continuations maintain pointers to their *parent* and *sibling* continuations. Thus, from any node within the DAG, the entire computation may be reconstructed.

Each continuation represents a task, i.e., a target, specification pair. The translation process is thus staightforward:

Begin by parsing the top level s-expression using the parse rules outlines in Chapter 3.2, page 13. This results in a parse-tree representation of the user's input where each leaf is either an atomic constituent, such as a number of symbol, or is a *simple expression* of the form:

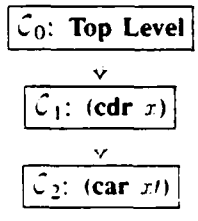
```
(Fn <arg-1> <arg-2> ... <arg-n>)
```

where each `<arg-i>` is atomic.

Simple expressions, such as funcalls with atomic arglists, can be represented as direct **requests**. All that is required, at this point, is to make sure that the references to the *parent* continuations exists before they are required to be compiled. For instance,

```
(car (cdr x))
```

will result in the chain of continuations shown in the following diagram:



Recall that each continuation, C_i, in this chain contains an environment object as well. Environments are built upon two layers:

a bindings layer which is a list of symbol-value pairs[3] which are local to this particular subform. For instance, bindings for x, in the previous example, are determined by looking x up in C_1's environment. This is the intention of the $x1$ in the next continuation box —to demonstrate that the reference to x in that case is not the same x as in the previous, hence the "1;" and,

[3] Known in Lisp parlance as an *association list*. Alternative representations could just as easily be chosen. Balanced trees, for instance, could be used for large environments. In the current implementation the default top level environment is a hash-table.

a parent layer which is a pointer to the previous environment. This is used to determine the context for the next reference.

Operationally, a reference to an identifier may be found locally, if available locally, or might be found in the current environment's parent environment, and so on. Eventually, the chain ends at the **Top Level** where either the identifier is found to be a "free reference" or bound in the global environment.

The structure of environments and the definition-use patterns of continuations all suggest a top down code generation approach —which is what the translation process currently employs. In summary, the DAG is walked, top-down, and a stream of VM instructions is generated.

4.2.12 Expanded Set of Canonical Types

Hence the migratability of data and ultimately TLC's general utility is dependent upon maintaining a rich set of data types: the more classes of data that TLC can manipulate, the greater its overall power.

But, the current set of canonical data types supported by CRONUS is limited. This set will need to be expanded as required as TLC evolves. At the present time there is a need to add a canonical representation of lists to the currently defined types, for instance.

4.2.13 Visualizing Concurrency

Since the specification of the earliest parallel programming languages, the problem of observing and debugging computations in parallel environments has assumed greater importance than it has received attention. The current TLC compiler utilizes the GRAPHER [Sus88] to help applications developers manipulate the DAG graphically. The GRAPHER provides a powerful tool for controlling the representional aspects of the problem. Figure 4.1 illustrates the DAG of a conditional expression[4] with concurrent then and else arms

```
(cIF (predicate a b)
  (then (then-fn-1 a) (then-fn-2 b) a)
  (else (else-fn-1 b) (else-fn-2 a) b))
```

as output by the compiler and graphed by the GRAPHER.

[4] Actually, a **cIF** expression, which was introduced in Chapter 3.3.3, page 20.

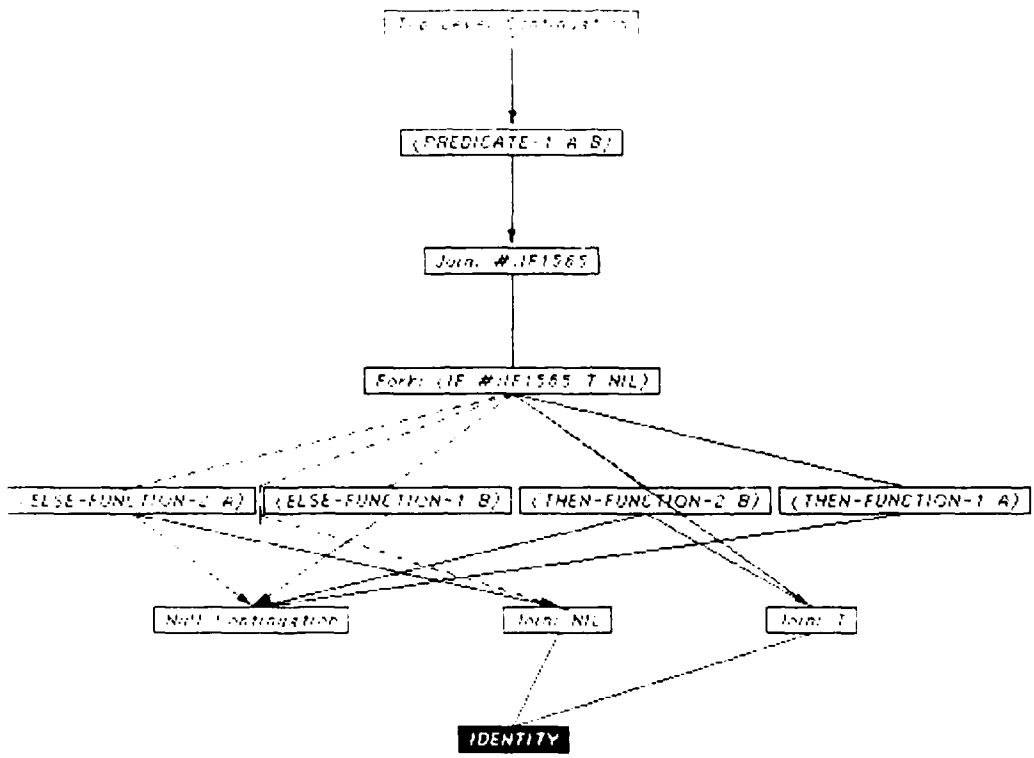


Figure 4.1: DAG of a conditional with concurrent Then and Else Clauses.

Graphical Programming Interface

Although there is no clear requirement for a graphical programming interface at this point, it may be desirable to investigate the attractiveness of this approach for particular user environments.

Viewing Computation at the VM Level

The TLC VM differs fundamentally from traditional machines. For one thing, its view of memory and locality is amorphous. Consequently, global data structures are eschewed for more localized, ephemeral structures that are as much an emergent property of the state of the network, coupled with the communications and resource allocation protocols in place at a particular moment, as the result of static definitions.

This decentralized perspective presents at once a challange and an opportunity to explore new directions in program definition, observation and debugging. Clearly, existing technology is inappropriate.

A promising direction is to *impose abstract ordering* over the *tasks*. For instance, *request* specifications can be paired with their *reply* analogs, as in the following (hypethetical) reconstruction of a recursive **Factorial**, $\mathcal{F}$, invoked with an integer argument, N. For simplicity's sake, assume that the message, $\mathcal{M}$, is a default method call that causes function to be invoked. In this abstract trace, we're calling each task invocation an *Event*, E, and each *request/reply* pair a *Transaction*, Tx.

The following event,

$$E_1 = \mathcal{F} \Leftarrow [\mathsf{Rq} : \mathcal{M}, \mathcal{C}_1, N]$$

is an abstract representation of the top level call: The integer, N, has been sent to $\mathcal{F}$ as a request. With the top level request is a continuation, $\mathcal{C}_1$, to which a reply will eventually be directed.[5]

The remaining request are now reconstructed,

$$E_2 = \mathcal{F} \Leftarrow [\mathsf{Rq} : \mathcal{M}, \mathcal{C}_2, N-1]$$
$$E_3 = \mathcal{F} \Leftarrow [\mathsf{Rq} : \mathcal{M}, \mathcal{C}_3, N-2]$$
$$\vdots$$
$$E_n = \mathcal{F} \Leftarrow [\mathsf{Rq} : \mathcal{M}, \mathcal{C}_n, 0]$$

and eventually the computation "bottoms out," at which point the actual computation begins. This is manifest as reply specifications, Rp, being sent to the appropriate continuations, the $\mathcal{C}_i$'s:

[5] For simplicity's sake the *resource-mgr* has been omitted.

$E_{n+1} = C_n \Longleftarrow [\mathsf{Rp} : 1]$
$E_{n+2} = C_{n-1} \Longleftarrow [\mathsf{Rp} : 2]$
$E_{n+3} = C_{n-2} \Longleftarrow [\mathsf{Rp} : 6]$
⋮
etc.

Alternately, the same computation could be viewed as a series of transactions,

$\mathsf{Tx}_1 = E_1, E_{n+3}$
$\mathsf{Tx}_2 = E_2, E_{n+2}$
$\mathsf{Tx}_3 = E_3, E_{n+1}$
⋮
etc.

which are recursive nestings of complementary request and reply pairs.

A graphical interface that organized and displayed these sets of ordered request/reply pairs, or transactions, would provide a good starting point for continuing research and development.

4.2.14 Apparent vs. Actual Grain Size

Although the TLC language and its virtual machine are designed with large grain, heterogeneous systems in mind, they are not *de facto* limited to such applications. With the addition of intrisincally finer grained primitives, e.g., **future** and **race** expressions, a finer grain of concurrency can be realized. These could be implemented at the primitive level as provided by the current version of TLC.

4.3 Major Enhancements

Numerous areas are important for longer term research related to TLC. They are identified and described briefly below:

4.3.1 Knowledge-Based Scheduling

As described in Chapter 3.3.5, the current TLC virtual machine does not apply any domain knowledge or run time status information in making resource allocation decisions. Adding such capability is currently viewed as perhaps the highest priority of the major enhancements.

4.3.2 Extensions to Functional Programming

Multiple Schedulers

The single scheduler inherent in the present design is likely to become a bottleneck if the system is heavily used. The ability to share the coordination of large numbers of tasks, i.e., *load balancing*, among mutiple schedulers is an important area of investigation. Some current work in this area has been undertaken by T. Malone em et al in the design of the ENTERPRISE system [MFGH88]. We feel that this work is appropriate and useful to our abstraction and are working to provide the mechanisms to support such bid and ask protocols.

Resource Managment

Resource control must be distributed with the consumer, i.e., the process that is requesting use of the resource. To this end, a *resource manager* mechanism, similar to *sponsors* as proposed by Manning, Hewitt, *et al* [Agh86], has been put in place.

Data Level Concurrency

Another type of concurrency that TLC capitalizes on is data concurrency. Recall the example query from Chapter 2.1, page 8:

```
(DefFunction CONCURRENT-SPASEARCH(a x z)
  (SPASEARCH (TIME-LATE
              (LAR-DISTANCE
               (POSITION x)
               (POSITION a)))
             (SPEED x)
             (SEARCH-RATE
              (SEARCH-SPEED x)
              (SWEEP
               (ENVIRONMENT-FACTOR a)
               (SENSOR-FIGURE-OF-MERIT x)
               (SIGNAL a)))
             Z))
```

Here, one must sequence over the elements in Displayed-Subs and select the submarine with the largest value of SPASEARCH. Clearly, response time can be improved by pipelining the processing of individual submarines rather than completely processing each submarine in turn. Near term developments include the specification of *streams* within TLC and generic operations on them. Thus, with the use of streams, data level pipelining is available: The system can be computing the LAR distance for submarine 2 at the same time it is computing the Time Late for submarine 1.

PRIMITIVE MODULES

MODULE NAME	MODULE INPUTS	MODULE OUTPUT	RUNS ON
SPASEARCH	Total-time Travel-time Search-rate	Probability-of-Detection	Butterfly
TIME-LATE	LAR-distance Max-speed	Time-late	Local Symbolics
LAR-DISTANCE	Position-1 Position-2	LAR-distance	FRESH Symbolics
SEARCH-RATE	Search-speed Sweep-width	Search-rate	Local Symbolics
SWEEP	Environment-factor Sensor-figure-of-merit SPA-signal-factor	Sweep-width	VAX

Figure 4.2: Decision Support System, Overview.

Primitives for Support Load Balancing

A TLC VM is a set of *dispatcher* objects, some defined locally, others distributed across the network of computational resources.

4.4 Integration with an Intelligent User Interface

Figure 4.2 illustrates a high level view of a typical decision support system. The module labeled User Command Interpreter accepts a command or query from the user and outputs a combination of subsystem operations that when executed properly will address the user request. The generation of this "program" depends on the User Command Interpreter's knowledge of subsystem functionality. This program can be used as an input to TLC.

The User Command Interpreter module incorporates the subsystems commonly referred to as the man-machine interface (MMI) and the command processor, both of which can be quite complex. For example, the MMI might include natural language understanding and the command processor could require a complex problem solver that in the most general case does program generation. Such systems are active areas of research.

In the near term, TLC will be used by a software developer to implement decision support system functionality. At run-time, the decision support system will retrieve and execute developer written programs based on user command input. Eventually, to the extent that program generation by the User Command Interpreter becomes available, one will be able to support a wider range of user commands and generate the programs that carry them out at run-time. Operation of the TLC is independent of whether it receives its input from a human or another software module.

5. Related Work

The work proposed here is related to work being carried out by a number of other groups. It is most closely related to ongoing work in distributed operating systems[STB86, BRS*85, JRT84]. MACH and Matchmaker[BRS*85, JRT84] have been used as the basis for both multiprocessor operating systems and distributed operating systems, however, the emphasis here has been on integrating systems based on Unix.

AGORA has been implemented on top of MACH to provide a shared memory virtual machine independent of the underlying hardware configuration.

Cronus[STB86] has been developed with support for heterogeneity a primary design goal. Because it runs under existing operating systems, it provides a basis for integration of existing as well as new application subsystems. Although it provides all of the rudimentary mechanisms that are needed to support concurrency, it does not integrate them into a set of high level language constructs that facilitate parallel programming. Distributed operating systems have begun to address some of the issues related to managing replicated resources, however, the schedulers associated with such systems have not incorporated the domain-related intelligence that we plan to eventually provide our intelligent scheduler.

Previous work on languages for parallel programming has had to address many of the same issues related to expressing concurrency that are of concern to us. Languages such as QLISP [GM88] and Multilisp [Hal86] have defined, implemented, and experimented with different language extensions and constructs. However, these languages have been oriented toward medium grain concurrency and have focused on procedural parallelism, not parallelism associated with pipelining of data. Dataflow languages [Ack82] have focused on pipelining of data but have been primarily associated with medium or fine grain parallelism. TLC draws heavily upon the work of Carl Manning [Man87] and the Message Passing Semantics Group at the Massachusetts Institute of Techonology. It differs from that effort, however, in its degree of integration, i.e., in TLC numbers are not message passing objects, and, again, in its assumptions about grainularity. Our work incorporates ideas developed by each of these groups to the extent that they apply to very coarse grain parallelism. In addition, we are also focusing on providing simple easy-to-understand as well as powerful constructs.

Work on next-generation software development environments is a third general area where work related to TLC exists. ABE [ELH87, HEF*88], a multi-machine development architecture for implementing intelligent systems, is perhaps the most closely related

work. ABE supports hierarchical modular programming where different types of control regimes called frameworks (dataflow, blackboard, transaction processing, and procedural) can be associated with subets of the individual modules. ABE supports the construction of (domain independent) skeletal systems which can be customized for particular applications. In contrast to TLC, however, ABE focuses on higher level issues and assumes that an appropriate distributed operating system subsubstrate exists.

Although not directly related to the proposed activities, there is very interesting work related to the more global problems of decision support systems going on at several organizations. Current work at BBN on the backend of the JANUS natural language system has emphasized decomposing a single user command into a composite command to multiple underlying systems. Similar work at ISI [PB88] has emphasized the command processing problem.

Appendix A. Underlying TLC Layers

A.1 TLC: At the Language Designer's Level

Much of the current compiler's work lies in translating user-level forms, such as **let** and **if**, into parse trees of **dlet** and **dif** forms.[1] Three forms, **defexpression**, **defcommand** and **defmacro** forms are provided for bootstrapping application level constructs into the compiler's primitives, i.e., **dlet** and **dif**. For instance, **let**, whose syntax mimics the COMMON LISP form by the same name, is defined as:

```
(DefExpression LET (let-arms let-body)
  `(dlet ,(mapcar #'(lambda (arm)
                      (let ((let-variable (car arm))
                            (let-exp (parse (cadr arm))))
                        `(,let-variable ,let-exp)))
                  let-arms)
     ,(parse let-body)))
```

which indicates that its semantics closely mimics that of **dlet**, described on page 42, below.

Sequential analogs for **let**, i.e., **let***, are available by treating each **let-arm** as a recursively nested **dlet** arm:

```
(let* ((v1 <exp-1>)
       (v2 <exp-2>)
       ...
       (vn <exp-n>))
  <body>)
```

is exactly equivalent to:

```
(dlet ((v1 <exp-1>))
  (dlet ((v2 <exp-2>))
    ...
     (dlet ((vn <exp-n>))
       <body>)))
```

[1] These are the basic forms out of which the compiler builds more complex behaviors. See A.3.1, page 42, below for more complete treatment.

Finally, the **cIF** form, which was defined as a macro in Chapter 3.3.3, page 20:

```
(DefExpression cIF (pred then-clauses else-clauses)
  (let ((then-value (parse (last then-clauses)))
        (else-value (parse (last else-clauses))))
    (parse
     `(if ,predicate
       (let ,@(mapcar #'(lambda (then-clause)
                          `(ignore ,(parse then-clause)))
                      (butlast then-clauses)))
       (let ,@(mapcar #'(lambda (else-clause)
                          `(ignore ,(parse else-clause)))
                      (butlast else-clauses)))))))
```

A.1.1 Sequencing Evaluation for Flow Of Control

Cases arise in which concurrency should be limited, as in "flow of control" problems. COMMON LISP, for instance, specifies such an order of evaluation for its boolean operations, **and** and **or**. Sequential analogs are also defined in TLC as compiler primitives. An example of one is **or***:

```
(DefExpression OR* (&rest subexpressions)
  `(if (null subexpressions)
       NIL
       (parse (if ,(car subexpressions)
                  T
                  (or* ,@(cdr subexpressions))))))
```

and* is straightforward and is omitted.

A.2 Present Shortcomings

From an operational perspective, however, both solutions, the simply concurrent case using **or** and more constrained, sequential case using **OR***, lack a certain elegance and descriptiveness. What is preferred is some way of saying: Evaluate all of the subexpressions concurrently. As soon as one returns a positive result, return that result and stiffle the on-going computation. In the present state of the language we have no way of expressing this.

A.3 TLC: At the Primitive Level

A.3.1 Internally Generated Forms

dlet and **dif** are the stuff out of which a spectrum of linguistic entities are formed; users do not program in terms of these —rather their code is translated into complex nestings

of these forms. Specifically, the syntax of **dlet** is:

```
DLET          ::    (DLET (<arm>*) <body>)
<arm>         ::    (<letvar> <expression>)
<letvar>      ::    <symbol>
<expression>  ::    DLET | DIF | <symbol> | <constant>
<constant>    ::    <number> | <quote-exp> | <symbol>
<quote-exp>   ::    '<expression>
<body>        ::    <expression> | <form>
<form>        ::    <expression>*
```

and its semantics are as follows: Each <arm> is evaluated concurrently, and each <form> comprising the body is evaluated concurrently. Evaluation of the <body> is postponed until the value for each <letvar> is available. Hence, a joining continuation is created at the border of each <body>. The value of a **dlet** expression is the value returned by the last <expression> as it appears in the program text.

Primitive flow of control is provided by **dif** forms. Its syntax is:

```
DIF           ::    (DIF <pred> <then-arms> <else-arms>)
<pred>        ::    <expression>
<then-arms>   ::    (then <form>)
<else-arms>   ::    (else <form>)
```

and its semantics are: <pred> is evaluated first. If <pred> returns a non-nil value, the <then-arms> are evaluated; the <else-arms> are evaluated otherwise. The evaluation of the arms entails the concurrent evaluation of the <expression>'s enclosed by the <form>'s. Without loss of generality, the then and else may be considered shorthand for **dlet**'s, as in:

```
(dif <pred>
     (then <exp-1> <exp-2> <exp-3>)
     (else <exp-1> <exp-2>))
```

is equivalent to

```
(if <pred>
    (let ((ignore <exp-1>)
          (ignore <exp-2>)
          (value <exp-3>))
      value)
  (let ((ignore <exp-1>)
        (value <exp-2>))
    value))
```

Bibliography

[Ack82] W.H Ackerman. Dataflow languages. *IEEE Computer*, 15–25, February 1982.

[Agh86] G. Agha. *Actors A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[Bac78] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[BH77] H. Baker and C. Hewitt. The Incremental Garbage Collection of Processes. In *Conference Record of the Conference on AI and Programming Languages*, pages 55–59, ACM, Rochester, New York, August 1977.

[BRS*85] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young. Mach-1: an operating system environment for large-scale multiprocessor applications. *IEEE Software*, July 1985.

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[ELH87] L.D. Erman, J.S. Lark, and F. Hayes-Roth. *ABE: An Environment for Engineering Intelligent Ssytems*. Technical Report TTR-ISE-87-106, Teknowledge Inc., November 1987. To appear in IEEE Transactions on Software Engineering, special issue on artificial intelligence.

[GM88] Richard P. Gabriel and John McCarthy. Qlisp. In J. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, Kluwer Academic Publishers, 1988.

[Hal86] Robert H. Halstead. Parallel symbolic computing. *IEEE Computer*, 35–43, August 1986.

[HEF*88] F. Hayes-Roth, L.D. Erman, S. Fouse, J.S. Lark, and J. Davidson. *ABE: A Cooperative Operating System and Development Environment*. Technical Report TTR-ISE-88-105, Teknowledge, Inc., May 1988. To appear in AI Tools and Techniques, Aug. 1988.

[JRT84] M.R. Jones, R.F. Rashid, and M.R. Thompson. *Matchmaker: An Interface Specification Language for Distributed Processing*. Technical Report CMU-CS-84-161, Carnegie-Mellon University, Dept. of Computer Science, December 1984.

[Man87] Carl R. Manning. *Acore: The Design of a Core Actor Language and its Compiler*. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1987.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3((4)):184–195, 1960.

[MFGH88] Thomas W. Malone, Richard E. Fikes, Kenneth R. Grant, and Michael T. Howard. Enterprise: A Market-like Task Scheduler for Distributed Computing Environments. In B.A. Huberman, editor, *The Ecology of Computation*, pages 177–205, Elsevier Science Publishers B.V. (North-Holland), 1988.

[PB88] Jasmina Pavlin and Raymond L. Bates. *SIMS: Single Interface to Multiple Systems*. ISI Research Report 88-200, ISI, February 1988.

[STB86] Richard E. Shantz, Robert H. Thomas, and Girome Bono. The architecture of the cronus distributed operating system. In *Proceedings of the Sixth International Conference in Distributed Computing Systems*, pages 250–259, Cambridge, Mass., June 1986.

[Ste84] Guy L. Steele Jr. *Common Lisp*. Digital Press, 1984.

[Sus88] J. Sussman. *The Grapher*. Technical Report 6876, BBN Systems and Technologies Corporation, July 1988.

[SVB*88] R. Schantz, S. Vinter, H. Barr, J. Berets, P. Bicknell, G. Bono, J. Bowe, J. Cole, M. Dean, C. Eide, R. Floyd, H. Forsdick, S. Jeffreys, R. Machey, P. Neves, R. Salz, and K. Schroder. *Cronus A Distributed Operating System: Summary of Technical Progress, 1986-1988*. Cronus Project Technical Report No. 8 6877, BBN Laboratories,Inc., July 1988.